CS152 Fall 2003 – David Patterson
Homework 2 Solutions
V1.1 – Updated 10/7/2003

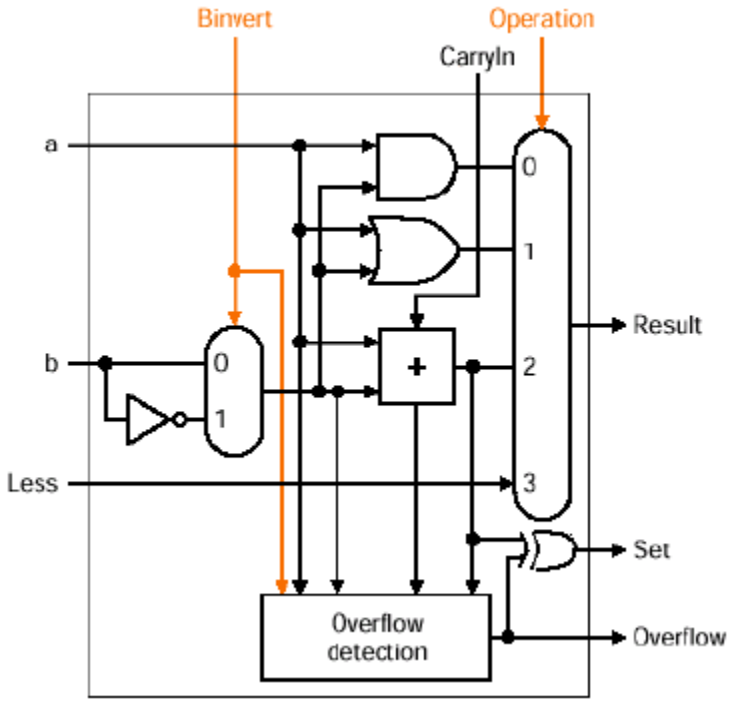
Send comments and corrections to Kurt.

10/7/2003: Finished the remaining problems. 6.7 probably has a typo.

4.13:

If the top bt of A_lower is a 0, no sign extension will occur, so A_upper_adjusted = A_upper. However, if the top bit of A_lower is a 1, then A_lower will be sign extended as an offset. This sign extension will place the value 0xFFFF in the upper bits of the 32-bit A_lower. Of course, 0xFFFF is just –1, so we need to add one to A_upper in order to cancel out the –1 from the 0xFFFF. The result: A_upper_adjusted = A_upper + 1.

4.23:



5.2:

Signal fault	Effect	Will break
RegDst=Constant 1	Mem writes always go to $rd	Lw (since it uses rt)
ALUSrc = Constant 1	2nd input to Alu is always immediate	Any R-format/beq
MemToReg = Constant 1	Result of Alu is always	Any R-format

	treated as a Mem pointer	
Zero = Constant 1	All branches are taken	Any branch

5.11:

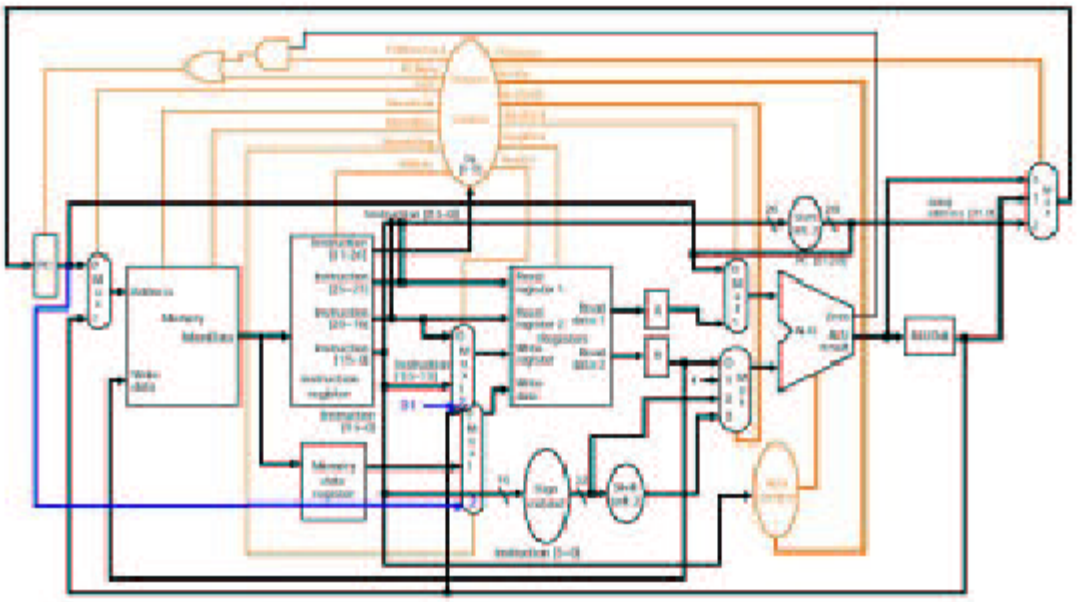
	BEQ	J	LW	SW	ADD	SUB	AND	OR	SLT
MemToReg	X	X	1	X	0	0	0	0	0
ALUSrc	0	X	1	1	0	0	0	0	0
RegDst	X	X	0	X	1	1	1	1	1
Branch	1	X	0	0	0	0	0	0	0

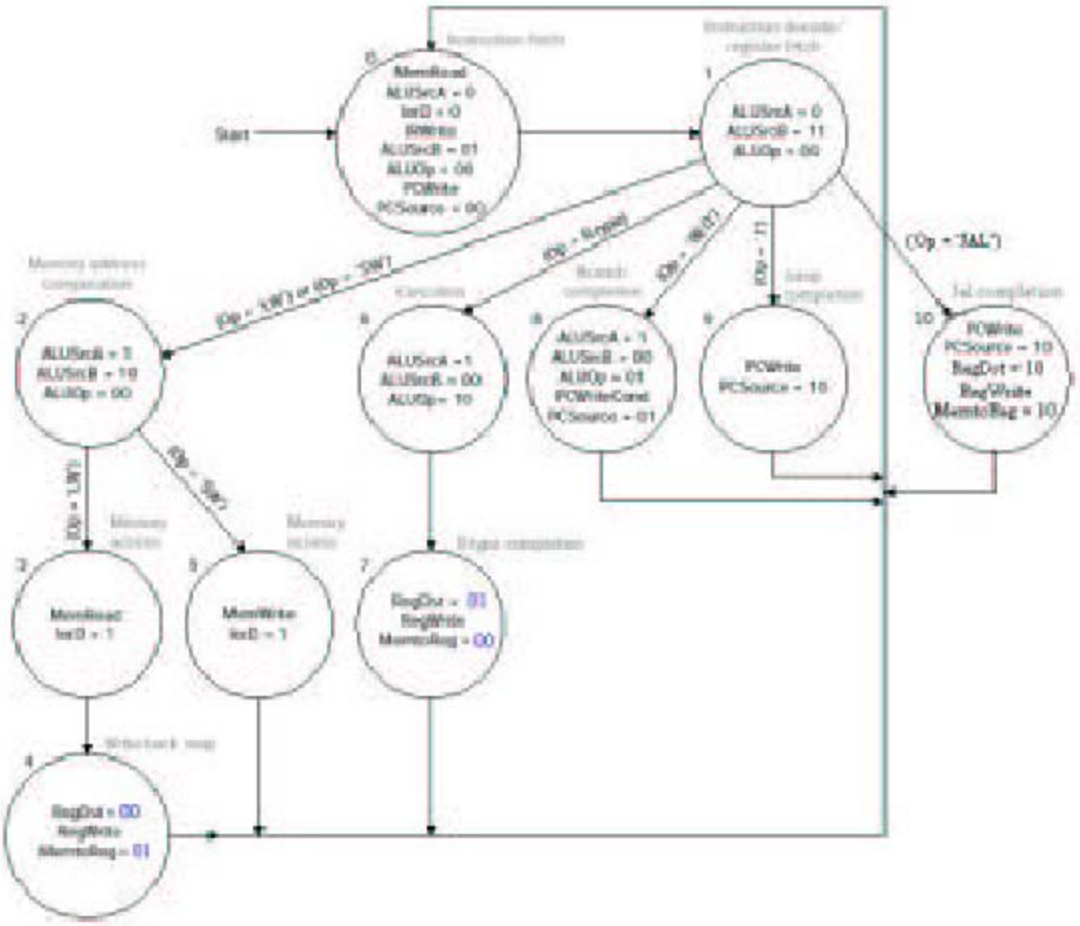
MemToReg = ALUSrc;
RegDst = NOT(AluSrc);

There are all sorts of other combinations ….

5.17:

Datapth additions in blue:





5:20:

The jump memory instruction behaves much like a load word until the memory is read. The data coming out of memory needs to be deposited into the PC. This will require a new input to the multiplexor controlled by PCSource. We add a new state coming off of state 3 that checks for Op = "jump memory" (and modify the transition to state 4 to ensure Op = "lw"). The new state has PCWrite, PCSource = 11. The transitions to states 2 and 3 need to also include Op = "jump memory."

5.27:

Block Copy - Multicyle & Microcode

Using the multicycle datapath from the textbook (Fig 5.33) changes are made below in aqua dashed lines. The basic RTL for the block copy instruction using this modified datapath is also referenced below. When instruction is fetched you have register $t1, $t2, and $t3. Make sure that the instruction register has a write enable so that the register file addresses are kept around. First you want to read R($t1) and R($t2) and get these values into the register for Reg1Data (register A) and Reg2Data (register B). Make sure these registers have write enables also. Next you want to use the value in register A to be used as address for memory to get the first value to be copied. At the same time you can use increment the register A value using the ALU. In the next cycle you can write the value back to memory using the memory address in register B. At the same you can write the newly incremented source address back into the register file, and use the ALU to increment the destination address in register B, and to get the length of the array into register A. In this final step you can write the value of incremented destination address back into the regfile. In the same cycle you can also decrement the length by one and check to see if it's negative. If so then block copy is done, if not it needs to write the length back into the register file and continue the loop. Another way is to store the length in it's own separate register (RegC, not shown in diagram), instead of ALUout, so that you won't need to write it back into the regfile. This saves you a cycle on every loop.
In hardware it would take 4cycles for the last loop + (5cycles)*(length -1) + 2cycles for instruction fetch and decode. The five cycles for the loop comes from having to store the length back into R($3). [If use extra register (RegC) to store length, instead of storing it to ALUout then to R($3); it's just (4cycles)*(length) + 2cycles.] In the software version it would take 3cycles for the beq + (length-1)*(24cycles). The 24 cycles are based on the multicycle state diagram (Fig 5.42) where lw is 5cycles, sw is 4cycles, each addi is 4 cycles, and branch is 3cycles. (i.e. 5+4+(4*3)+3)

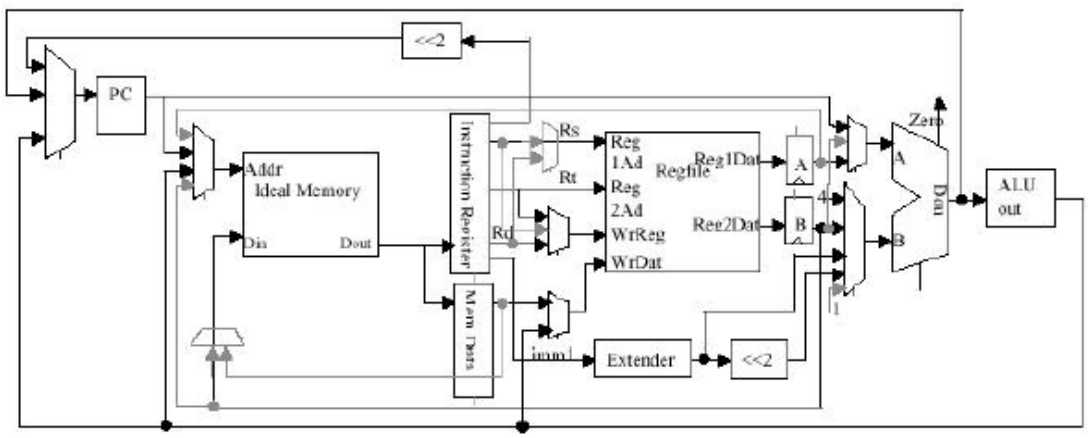
Basic RTL for Block Copy:

regA <- R($t1), regB <- R($t2)

MemDataReg <- M(regA), ALUout <- regA + 4

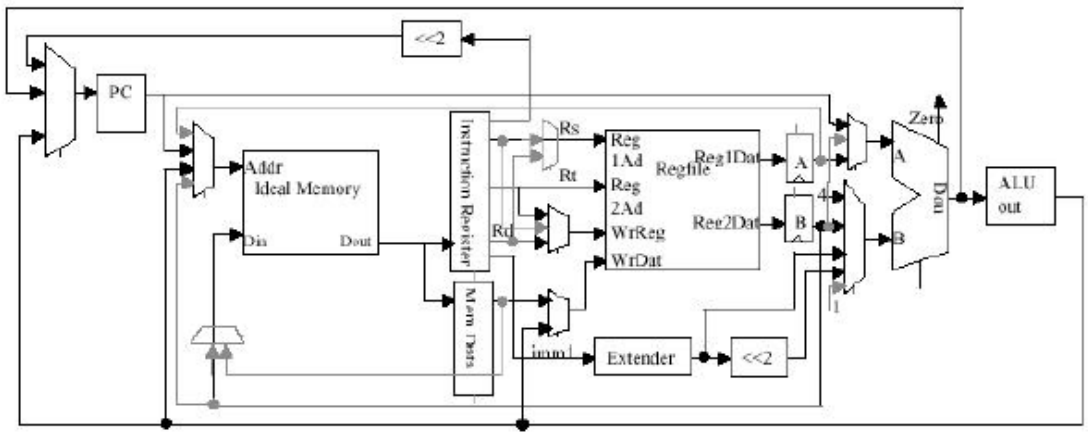
Mem <- MemDataReg at M(regB), R($t1) <- ALUout, ALUout <- regB + 4, regA <- R($t3)

R($t2) <- ALUout, ALUout <- regA - 1, decide if should loop again



5.28:

Mem <- MemDataReg at M(regB), R($t1) <- ALUout, ALUout <- regB + 4, regA <- R($t3)
R($t2) <- ALUout, ALUout <- regA - 1, decide if should loop again



6.2:

```
        add     $2, $3,   $4
        add     $4, $5,   $6
        add     $5, $3,   $4
```

6.3:

```
        addi   $3,$3,4
loop:   lw     $2, 96($3)
        beq    $3,$4,loop
        addi   $3,$3,4
```

6.7:

Cycle 5: PC=516:

IFID.Inst = (or $13 $6 $7)
IFID.NextPC = 516

IDEX.NextPC = 512
IDEX.Regwrite = 1
IDEX.MemToReg = 0
IDEX.Branch = 0
IDEX.MemRead = 0
IDEX.MemWrite = 0
IDEX.RegDst = 1
IDEX.ALUOp = 10b2
IDEX.ALUSrc = 0

IDEX.A = 14
IDEX.B = 15
IDEX.signImm = 0x00006024
IDEX.RegRT = 5
IDEX.RegRD = 12

EXMEM.NextPC = 508 + (0x00005822 *4)
EXMEM.RegWrite = 1
EXMEM.MemToReg = 0
EXMEM.Branch = 0
EXMEM.MemRead = 0
EXMEM.MemWrite = 0
EXMEM.ALUOut = -1
EXMEM.Zero = 0
EXMEM.WriteData = 13
EXMEM.RegDest = 11

MEMWB.NextPC = 584
MEMWB.RegWrite = 1
MEMWB.MemToReg= 1
MEMWB.ReadData = 1031
MEMWB.ALUOut = 31
MEMWB.RegDst = 10

6.27:

Branch should be resolved in ID stage. Forward the results right before EX/MEM and MEM/WB pipeline registers to ID stage.

Question 1:

```
sw      $7,     100($2)
lw      $8,     100($2)
```

no hazard for this case for the Mem write is commited in MEM stage not in WB stage while the the instrucitons involving registers commit results in the WB stage, which makes the forwarding necessary.

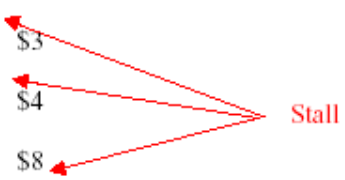
Question 2:

```
lw      $2, 100($5)
sw      $2, 100($6)
```

forwarding can be done as following:
If (MEM/WB.RegWrite and (MEM/WB.RegisterRd!=0)
And (MEM/WB. RegisterRd= = EX/MEM.RegisterRt)) Forward=01

6? 23

```
lw      $3,     0($5)
add     $7,     $7,     $3
lw      $4,     4($5)
add     $8      $8,     $4
sw      $6,     0($5)
add     $10,    $7,     $8
beq     $10,    $11,    Loop
```



6.26

```
        beq     $1,     $2,     target
        lw      $3,     40($4)
        add     $3,     $3,     $3
        sw      $3,     40($4)
target: or      $10,    $11,    $12
```

it is possible to have stall and flush simultaneously. Since in the pipeline fig6.51, the only stall source will be in ID stage, it is ok to stall pipeline in the IF and ID stage while flush later stages.